Construction and Verification of Software

2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 5 - State Change and Type States

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Outline

- Framing in Hoare Logic
- Modifying an Array
- ADTs and TypeStates
- Building a Queue with TypeStates

Construction and Verification of Software

2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 5 - Part I - Changing State

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

- This is captured by the derived (constancy or frame) rule

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}} \text{ where } M(P) \cap V(C) = \emptyset$$

- Provided that the variables modified by P (M(P)) are not referred by C (V(C)).

$$\frac{\{x > 0\}\ y := x\ \{y > 0 \wedge x = y\}}{\{x > 0 \wedge z < 0\}\ y := x\ \{y > 0 \wedge x = y \wedge z < 0\}}$$

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

- This is captured by the derived (constancy or frame) rule

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}} \text{ where } M(P) \cap V(C) = \emptyset$$

- Provided that the variables modified by P (M(P)) are not referred by C (V(C)).

- Updates to variables do not allow framing the modified variables.

```
method deposit(v:int)
  modifies this`bal
```

like one assignment

Changing State

- Tracking changes with dynamic memory is not covered by the original Hoare Logic. Each tool adopts some kind of strategy to make the frame rule sound.

```
function AbsInv():bool
  reads this`a, this`size, this`s, this.a
{ RepInv() && Sound() }

method add(x:int)
  requires AbsInv()
  ensures AbsInv()
  modifies this`a, this.a, this`size, this`s
```

memory referred by

field of

contents of

memory modified by

- Dafny refers to allocated memory areas. Objects and arrays. Modification of fields are modifications to the container object.

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}}$$

- Information in the interface is important to know modified and referred memory.

```
method Main() {
  var a:Account := new Account();
  a.deposit(10);
  a.withdraw(20); <<<<< ????
  if a.getBalance() > 10
  { a.withdraw(10); a.deposit(10); }
}
```

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}}$$

- Information in the interface is important to know modified and referred memory.

```
method Main() {
  var a:Account := new Account(); { a.bal >= 0 }
  { a.bal >= 0 } a.deposit(10); { a.bal >= 0 }
  { a.bal >= 0 } a.withdraw(20); <<<<< ????
  if a.getBalance() > 10
  { { a.bal > 10 } a.withdraw(10); a.deposit(10); }
}
```

Construction and Verification of Software

2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 5 - Part II - Changing an Array

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Modifying an array

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

```
method selectionSort(a:array<char>, n:int)
  requires 0 <= n <= a.Length
  modifies this.a
{
  var i := 0;
  while i < n
    decreases n - i
    invariant 0 <= i <= n
    invariant sorted(a, i)
    invariant partitioned(a, i, n)
  {
    selectSmaller(a, i, n);
    i := i + 1;
  }
}
```

```
method selectSmaller(a:array<char>, i
  requires 0 <= i < n <= a.Length
  requires sorted(a, i)
  requires partitioned(a, i, n)
  modifies this.a
  ensures sorted(a, i+1)
  ensures partitioned(a, i+1, n)
```

- It is like an assignment to all positions in the array

Modifying an array

- To have control about the positions that are indeed modified extra information is needed.

```
method sortedInsertion(a:array<char>, na:int, e:char)
returns (z:array<char>, nz:int, pos:int)
    requires 0 <= na < a.Length
    requires sorted(a, na)
    modifies this.a
    ensures sorted(a, na+1)
    ensures 0 <= pos < na+1 && a[pos] == e
    ensures forall k :: 0 <= k < pos ==> a[k] == old(a[k])
    ensures forall k :: pos < k < na+1 ==> a[k] == old(a[k-1])
```

- This allows the caller context to maintain (frame) knowledge about the unmodified positions.

- All knowledge about the array must be given by the post conditions.

Another example

- The Set example

```
method add(x:int)
requires AbsInv() && size() < maxsixe()
ensures AbsInv()
ensures s == old(s) + {x}
modifies this`s, this`size, this`a, this.a
{
   var idx := find(x);
    if( idx < 0 ) {
        a[size] := x;
        size := size + 1;
        s := s + {x};
   }
}
```

- The precondition about the maximum size is not nice…

Another example

- A more general implementation is needed to eliminate it

```
method add(x:int)
requires AbsInv()
ensures AbsInv()
ensures s == old(s) + {x}
modifies this`s, this`size, this`a, this.a
{
    if( size == a.Length ) { Grow(); }

    var idx := find(x);
    if( idx < 0 ) {
        a[size] := x;
        size := size + 1;
        s := s + {x};
        assert forall i :: 0 <= i < size-1 ==> a[i] == old(a[i]);
    }
}
```

Set ADT (growable)

```
 class ASet {
    // Abstract state
    ghost var s:set<int>;
    // Representation state
    var a:array<int>;
    var size:int;
…
    method Grow() returns (na:array<int>)
        requires RepInv()
        ensures size < na.Length
        ensures fresh(na)
        ensures forall k::(0<=k<size) ==> na[k] == a[k];
    {
        na := new int[a.Length*2];
        var i := 0;
        while (i<size)
            decreases size-i
            invariant 0 <= i <= size ;
            invariant forall k::(0<=k<i) ==> na[k] == a[k];
        {
            na[i] := a[i];
            i := i + 1;
        }
    }
```

memory not tracked before

Set ADT (growable)

```
class ASet {
    // Abstract state
    ghost var s:set<int>;
    // Representation state
    var a:array<int>;
    var size:int;
…
    method add(x:int)
    requires AbsInv()
    ensures AbsInv()
    ensures s == old(s) + {x}
    modifies this`s, this`size, this`a, this.a
    {
        if( size == a.Length ) { Grow(); }

        var idx := find(x);
        if( idx < 0 ) {
            a[size] := x;
            size := size + 1;
            s := s + {x};
            assert forall i :: 0 <= i < size-1 ==> a[i] == old(a[i]);
        }
    }
```

Set ADT (growable)

```
class ASet {
    method del(x:int)
        modifies this, a;
        requires AbsInv()
        ensures RepInv()
        ensures x in old(s) <==> old(s) == s + {x}
    {
      var i:int := find(x);
      if i >= 0 {
        var pos := i;
        while (i < size-1)
          modifies a;
          decreases size - 1 - i
          invariant pos <= i <= size-1
          invariant Unique(a,0,i) && Unique(a,i+1,size)
          invariant forall j::(0 <= j < pos) ==> a[j] == old(a[j])
          invariant forall j::(pos <= j < i) ==> a[j] == old(a[j+1])
          invariant forall j::(i+1 <= j < size) ==> a[j] == old(a[j])
        {
            a[i] := a[i+1];
            i := i + 1;
        }
        size := size - 1;
      }
```

Further hints on invariants

- We illustrate a famous issue related to using formal logic to reason about dynamical systems, the so-called “**frame-problem**”.
- There is no “purely logical” way of inferring what does not change after an action, we need in each case to specify for each action not only what changes, but also what has not (remains stable).
- E.g. this arises in reasoning about programs

```
{x.val() == a && y.val() == 0} x.inc() { x.val() == a+1 && y.val() == ?}
```

- How do we know changing x affects y or not?

Key Points

- The ADT operations pre / post conditions must always preserve the representation invariant
- Other operations (private helper methods) do not need to preserve the invariant, they are need to know about the ADT implementation details
- The ADT pre / post conditions should avoid referring to the concrete state, to preserve information hiding
- To do that, you may expose ghost variables
- Alternatively, use also some form of **typestate**, enough to express rich dynamic constraints (next).

Construction and Verification of Software

2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

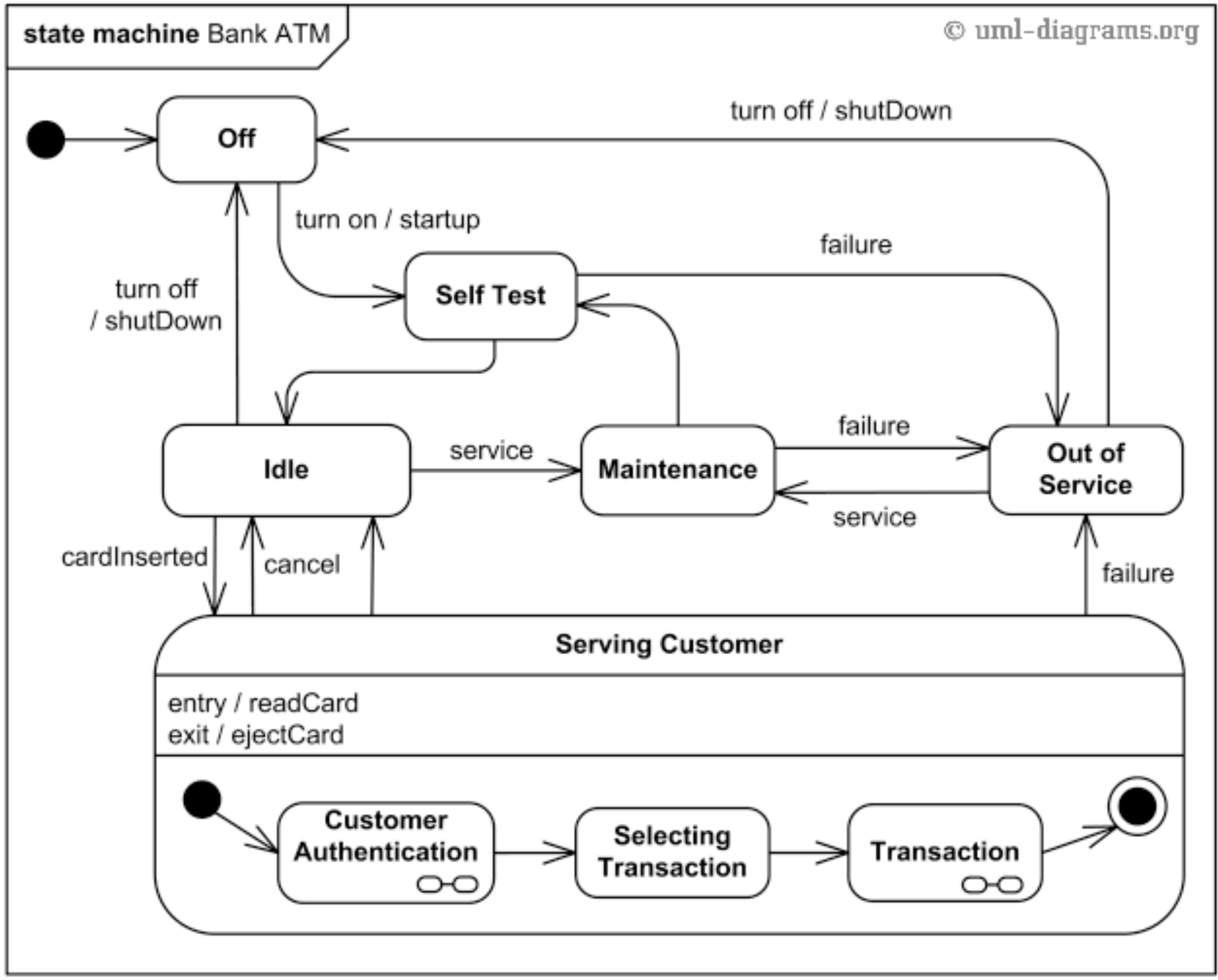
Lecture 5 - Part III - ADTs and Type States

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

UML State Transition Diagrams

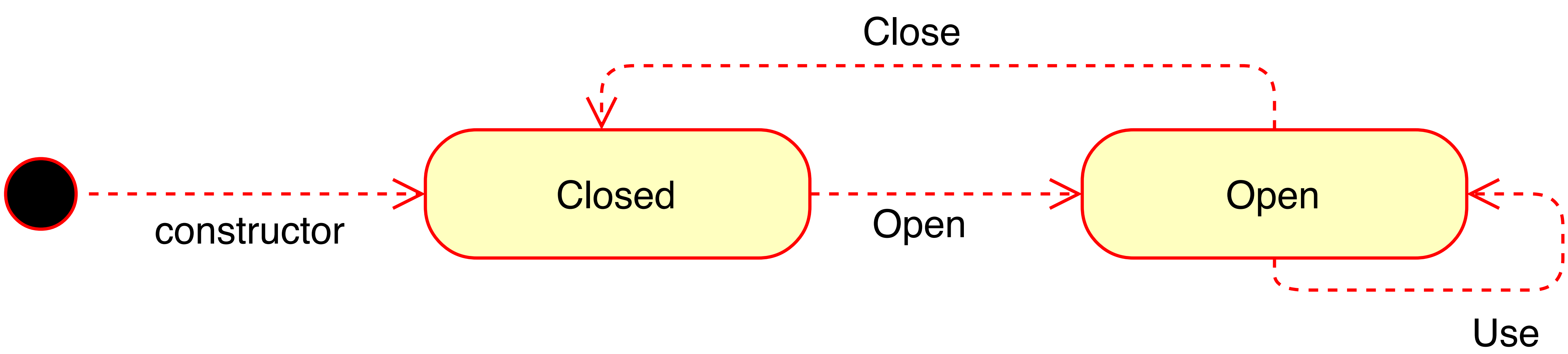
- Typically the connection between a state and the domain of the values for an object are based on conventions / written in documentations.
- Operations are state transitions in a state diagram.
- If a state is formally connected to conditions over the state of an object, the correction of state transitions may be mechanically checked



https://www.uml-diagrams.org/bank-atm-uml-state-machine-diagram-example.html?context=stm-examples

TypeStates

- In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states
- We illustrate using a general Resource object with the following state diagram



TypeStates

- In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states
- We illustrate using a general Resource object.
  - A Resource must first be created and starts on the closed state
  - A Resource can only be used after being Opened
  - A Resource may be Closed at any time
  - A Resource can only be Opened if it is in the Closed state, and Closed if it is in the Open state
- We define two abstract states (`ClosedState()` and `OpenState()`)

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { … }

    function ClosedState():bool
    reads this
    { …  }

    constructor ()
    ensures  ClosedState();
    { … }

  …

}
```

TypeStates define an abstract layer, visible to clients that can be used to verify resource usage.

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { … }

    function ClosedState():bool
    reads this
    { …  }

    constructor (
    ensures  Clos
    { … }

  …

}
```

```
method UsingTheResource()
{
    var r:Resource := new Resource();
    r.Open(2);
    r.Use(2);
    r.Use(9);
    r.Close();
}
```

Legal usage of resource, according to protocol!

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { … }

    function Clos
    reads this
    { …  }

    constructor (
    ensures  Clos
    { … }

  …

}
```

```
method UsingTheResource()
{
    var r:Resource := new Resource();
    r.Close();
    r.Open(2);
    r.Use(2);
    r.Use(9);
    r.Close();
    r.Use(2);
}
```

Illegal usage of resource, according to protocol!

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { h != null && 0 < size == h.Length }

    function ClosedState():bool
    reads this
    { h == null && 0 == size  }

    constructor ()
    ensures  ClosedState();
    { h := null; size := 0; }

  …

}
```

TypeStates define an abstract layer, that may be defined with relation to the representation type (and invariants) and be used to verify the implementation.

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;
 …
    method Open(N:int)
    modifies this
    requires ClosedState() && N > 0
    ensures  OpenState() && fresh(h)
    {
        h, size := new int[N], N;
    }

    method Close()
    modifies this
    requires OpenState()
    ensures  ClosedState()
    {
        h, size :=null, 0;
    }
```

Method Implementations represent state transitions, and must be implemented to correctly ensure the soundness of the arrival state (assuming the departure state)

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;
 …

    method Use(K:int)
    modifies h;
    requires OpenState();
    ensures  OpenState();
    {
       h[0] := K;
    }

 …

}
```

No execution errors are caused by misusing the representation type. Notice that states are `RepInv()` variants, essential to execute different method.

TypeStates

- In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states
- It is often enough to expose TypeState assertions to ensure ADT soundness and no runtime errors
- In general, full functional specifications in terms the abstract state is too expensive and should be only adopted in high assurance code
- However, TypeState assertions are feasible and should be enforced in all ADTs:
- The simplest TypeState is the RepInv (no variants/less specific).

Key Points

- Software Design Time
  - Abstract Data Type
  - What are the Abstract States / Concrete States?
  - What is the Representation Invariant?
  - What is the Abstraction Mapping?
- Software Construction Time
  - Make sure constructor establishes the RepInv
  - Make sure all operations preserve the RepInv
    - they may assume the RepInv
    - they may require extra pre-conditions (e.g. on op args)
    - they may enforce extra post-conditions
  - Use assertions to make sure your ADT is sound

Construction and Verification of Software

2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

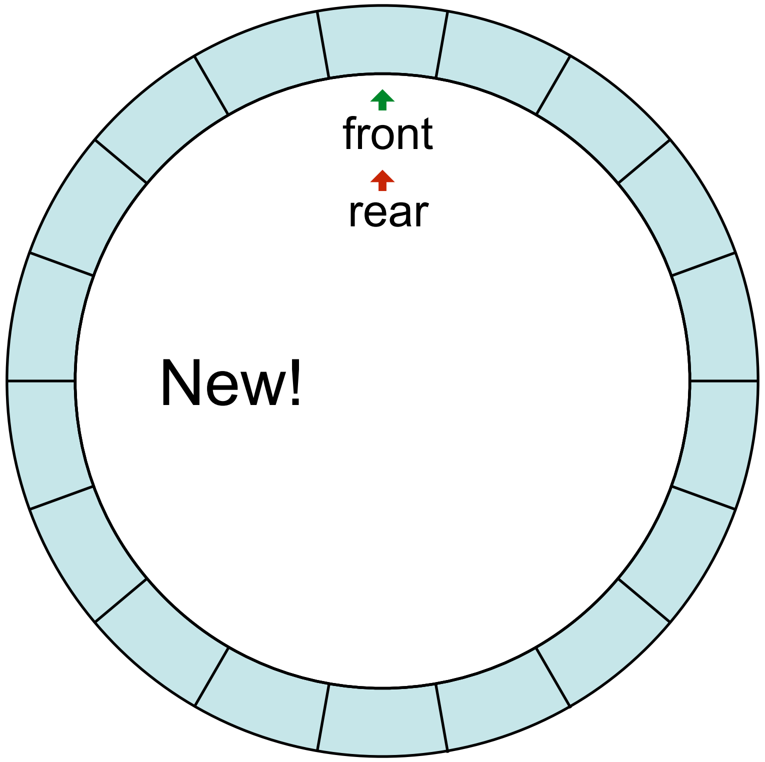
Lecture 5 - Part IV - Type States, an example

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

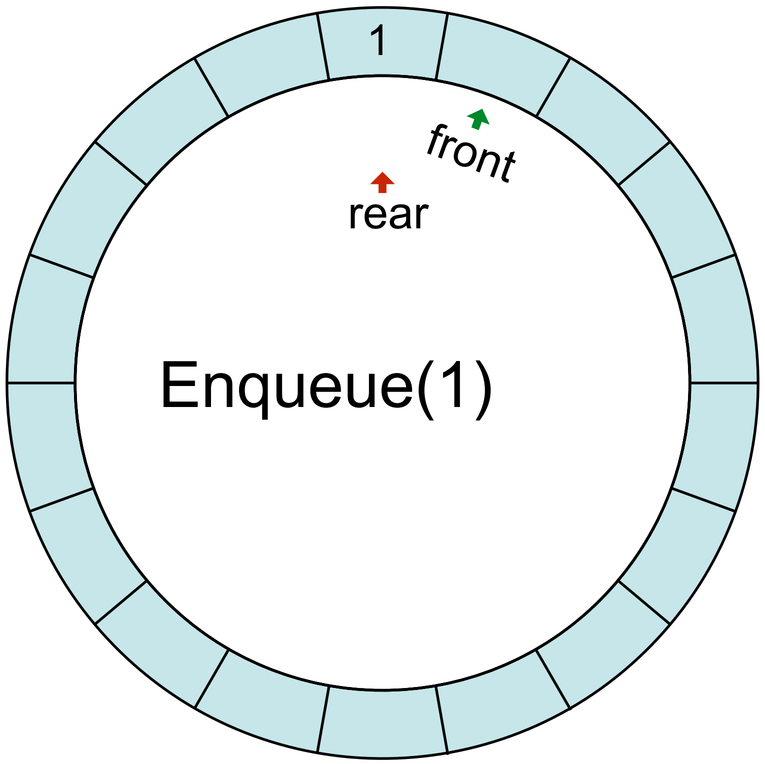
TypeStates - Queue

- An implementation using a circular buffer…



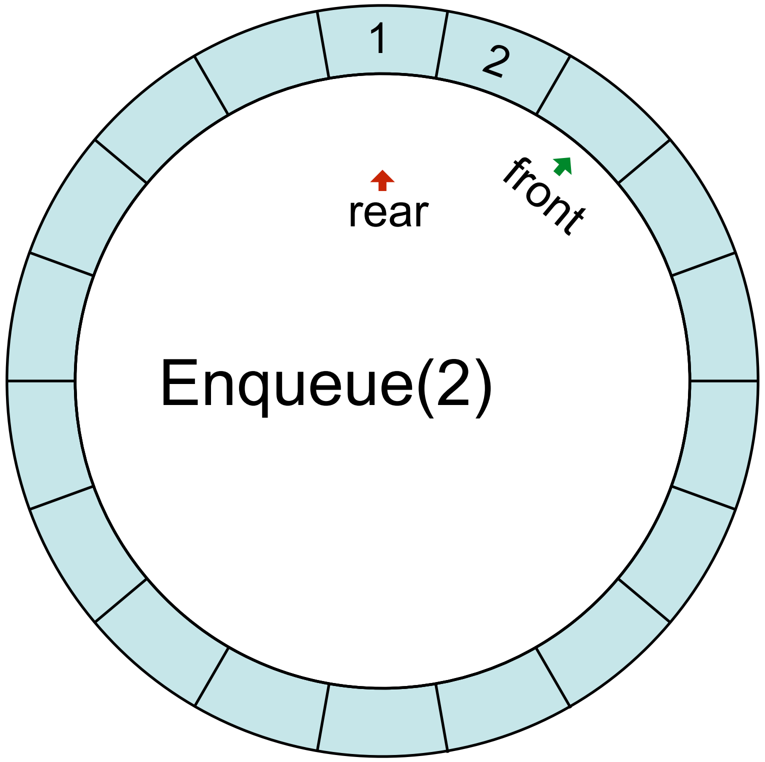
TypeStates - Queue

- An implementation using a circular buffer…



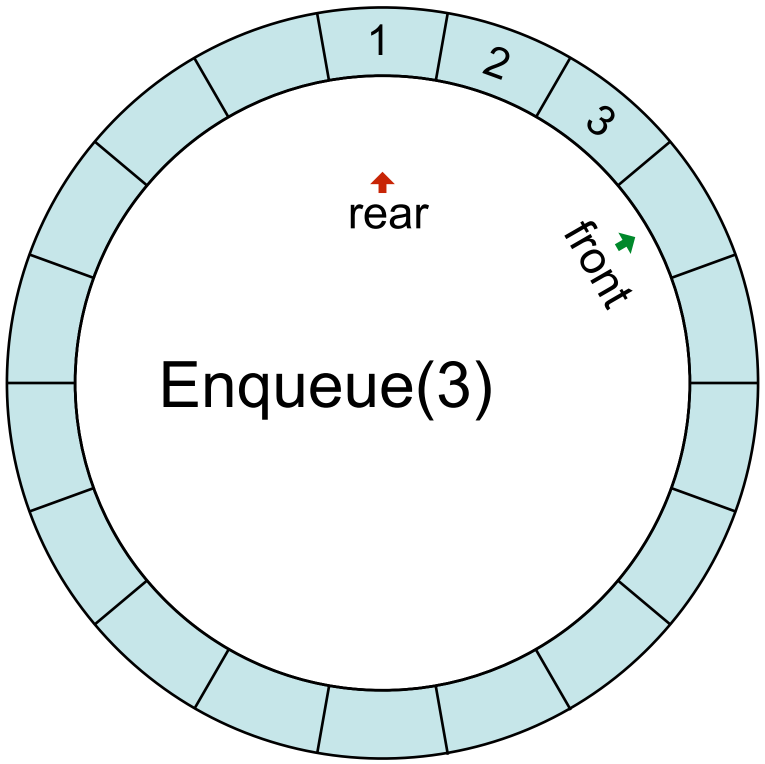
TypeStates - Queue

- An implementation using a circular buffer…



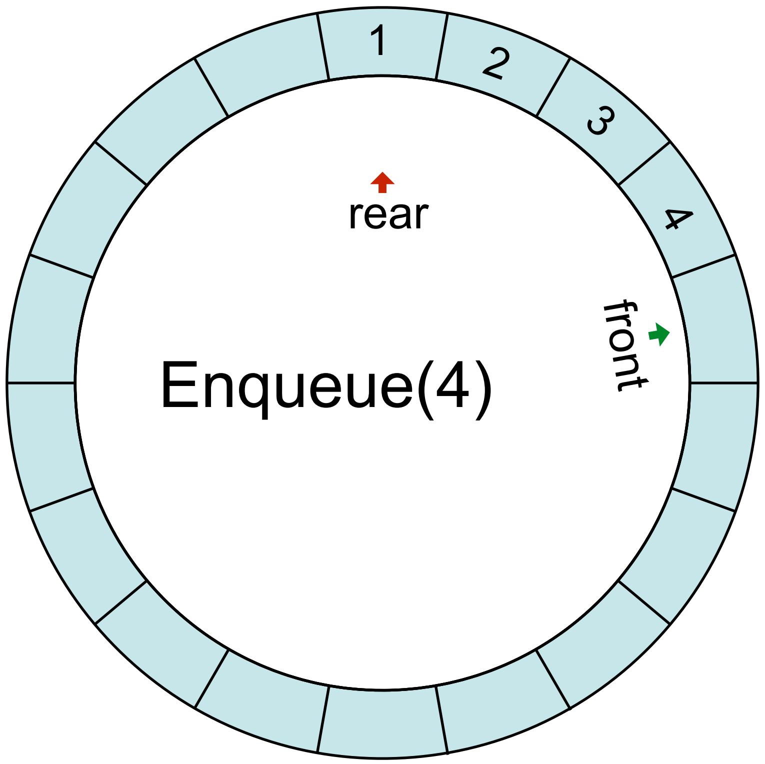
TypeStates - Queue

- An implementation using a circular buffer…



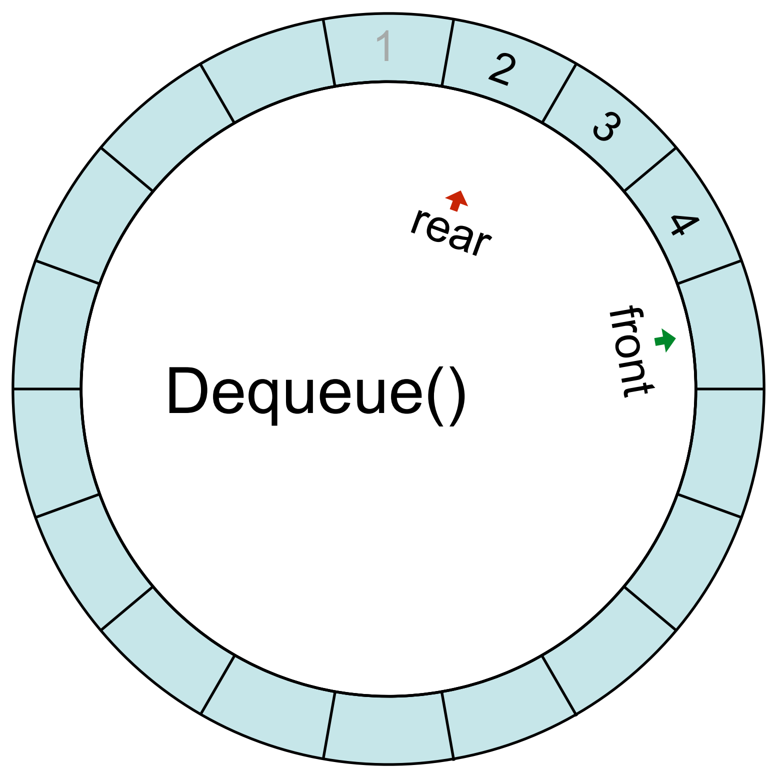
TypeStates - Queue

- An implementation using a circular buffer…



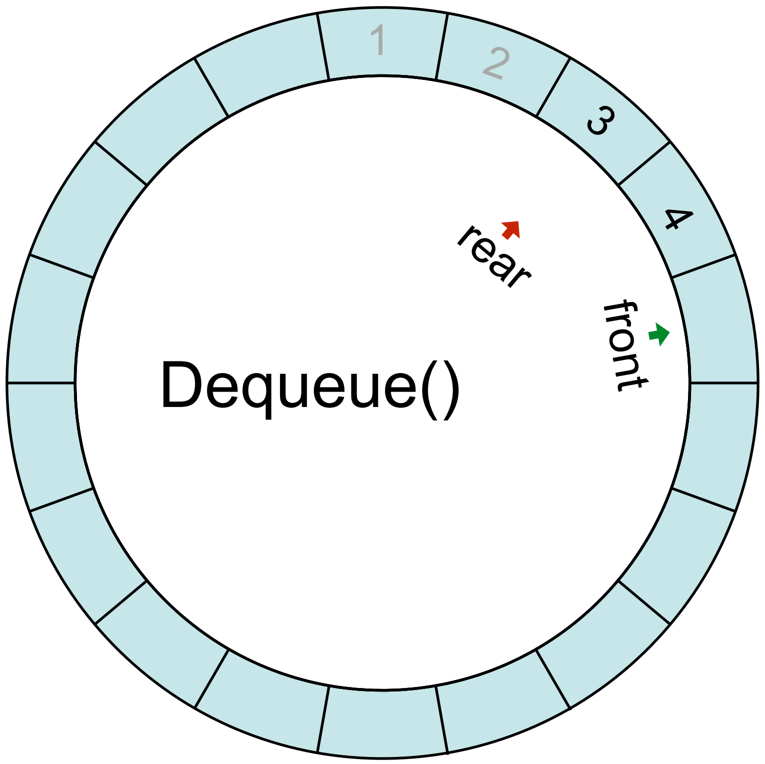
TypeStates - Queue

- An implementation using a circular buffer…



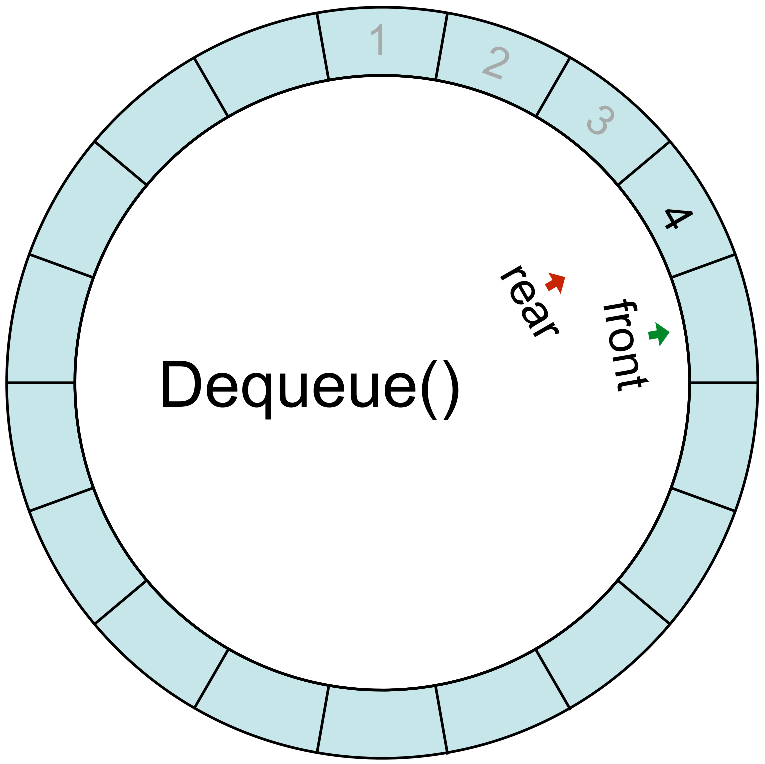
TypeStates - Queue

- An implementation using a circular buffer…



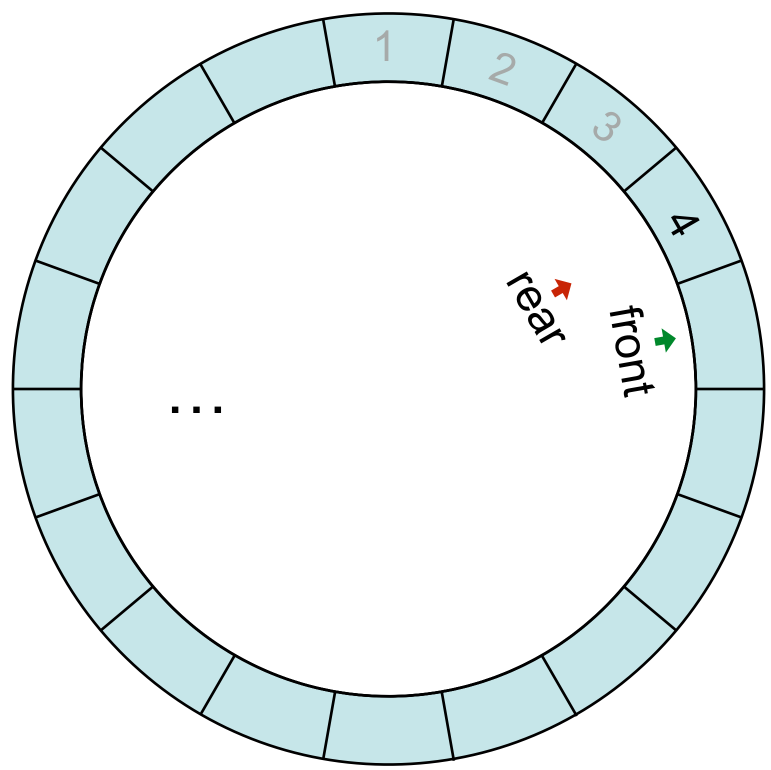
TypeStates - Queue

- An implementation using a circular buffer…



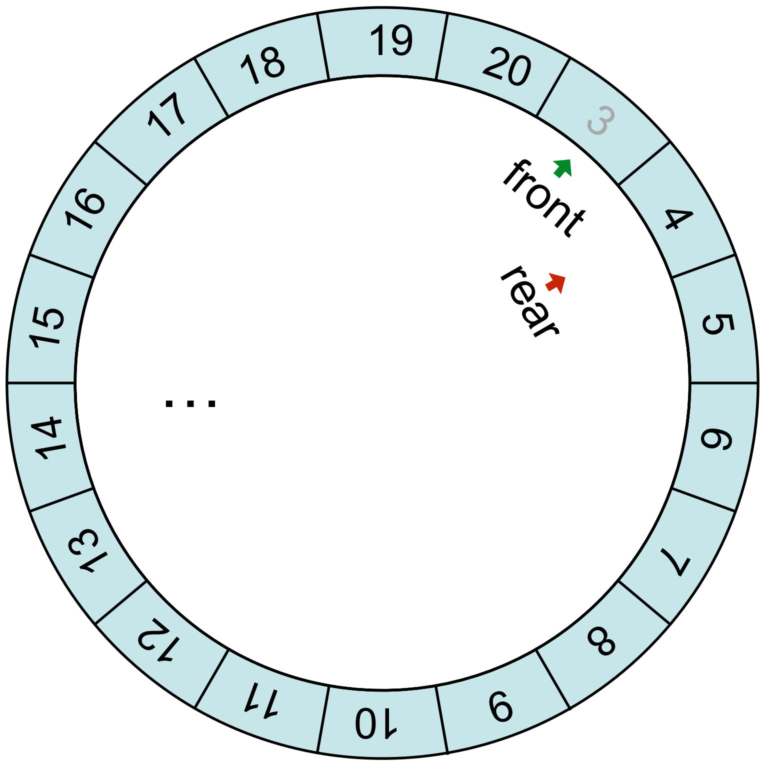
TypeStates - Queue

- An implementation using a circular buffer…



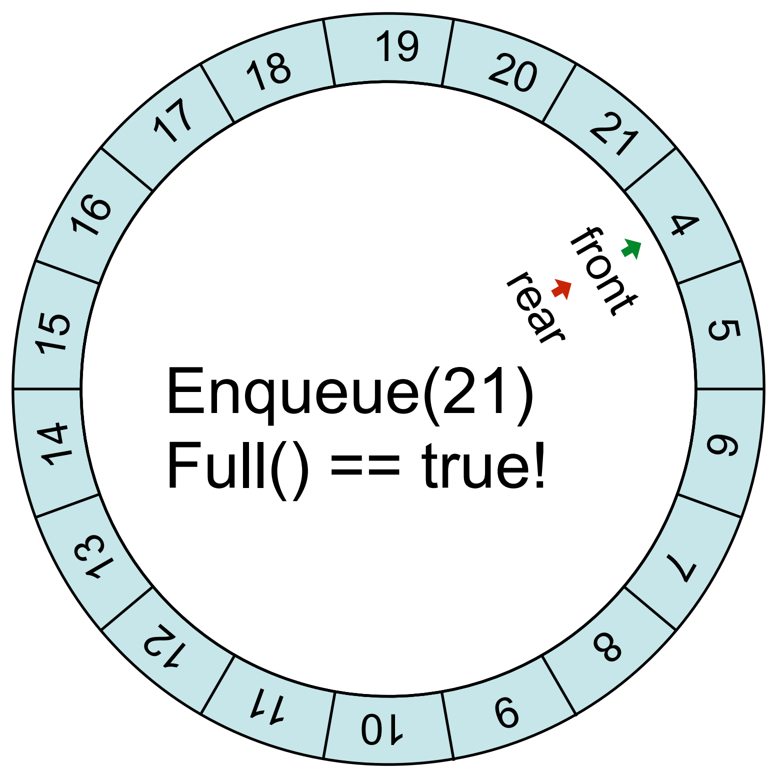
TypeStates - Queue

- An implementation using a circular buffer…



TypeStates - Queue

- An implementation using a circular buffer…



TypeStates - Queue

- An implementation using a circular buffer…

```
class Queue {
    // Representation type
    var a:array<int>;
    var front: int;
    var rear: int;
    var numberOfElements: int;

    // Representation invariant
    constructor(N:int)
        requires 0 < N
        ensures fresh(a)
    {
        a := new int[N];
        front := 0;
        rear := 0;
        numberOfElements := 0;
    }
    …
}
```

TypeStates - Queue

- What's wrong with it? a RepInv is necessary to maintain front and rear within bounds…

```
class Queue {

    …
    method Enqueue(V:int)
        modifies this`front, this`numberOfElements, a
    {
        a[front] := V;
        front := (front + 1)%a.Length;
        numberOfElements := numberOfElements + 1;
    }

    method Dequeue() returns (V:int)
        modifies this`rear, this`numberOfElements, a
    {
        V := a[rear];
        rear := (rear + 1)%a.Length;
        numberOfElements := numberOfElements - 1;
    }
}
```

TypeStates - Queue

```
class Queue {
    // Representation type
    var a:array<int>;
    var front: int;
    var rear: int;
    var numberOfElements: int;

    // Representation invariant
    function RepInv():bool
        reads this
    { 0 <= front < a.Length && 0 <= rear < a.Length }

    constructor(N:int)
        requires 0 < N
        ensures RepInv()
        ensures fresh(a)
    {
        a := new int[N];
        front := 0;
        rear := 0;
        numberOfElements := 0;
    }
    …
}
```

TypeStates - Queue

```
class Queue {
    …
    method Enqueue(V:int)
        modifies this`front, this`numberOfElements, a
        requires RepInv()
        ensures RepInv()
    {
        a[front] := V;
        front := (front + 1)%a.Length;
        numberOfElements := numberOfElements + 1;
    }

    method Dequeue() returns (V:int)
        modifies this`rear, this`numberOfElements, a
        requires RepInv()
        ensures RepInv()
    {
        V := a[rear];
        rear := (rear + 1)%a.Length;
        numberOfElements := numberOfElements - 1;
    }
}
```

TypeStates - Queue

- Not enough… No runtime errors but the correct behaviour is not yet ensured…
  wrong values may be returned,
  valid elements maybe overwritten… right?

```
method Main()
{
    var q:Queue := new Queue(4);
    var r:int;

    q.Enqueue(1);
    r := q.Dequeue();
    r := q.Dequeue();      ????
    q.Enqueue(2);
    q.Enqueue(3);
    q.Enqueue(4);
    q.Enqueue(4);    ????
    q.Enqueue(4);
    q.Enqueue(5);
}
```

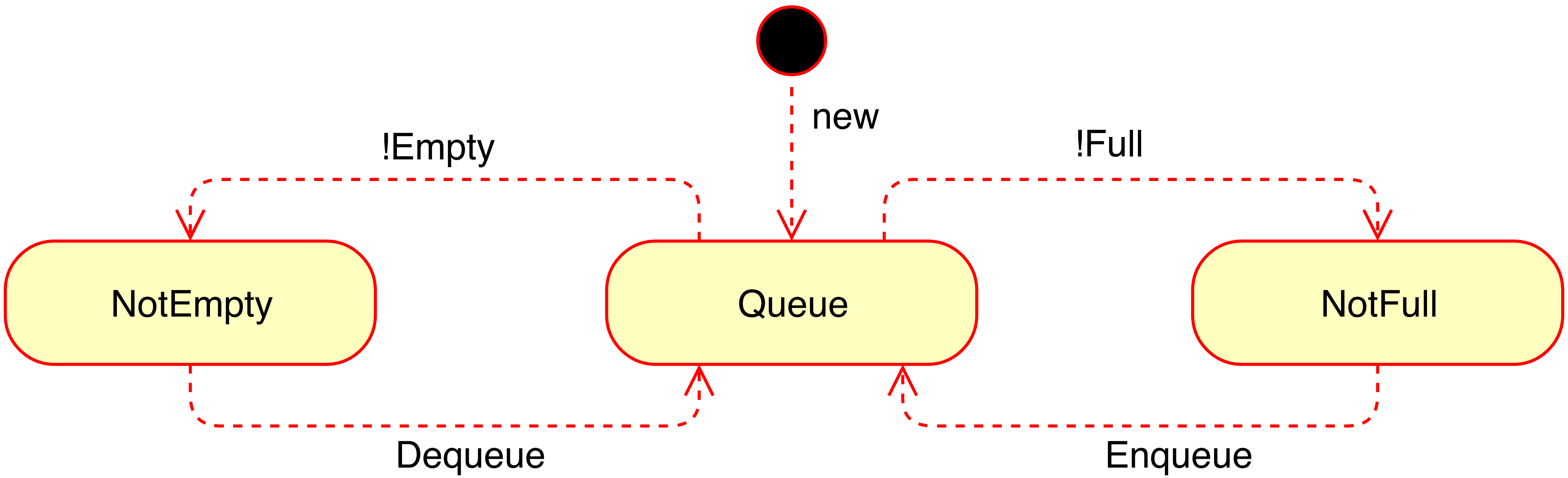
TypeStates - Queue

- RepInv must be refined to ensure that we stay inside the domain of valid queues…

```
function RepInv():bool
     reads this
 {
     0 <= front < a.Length &&
     0 <= rear < a.Length &&
     if front == rear then
       numberOfElements == 0 ||
       numberOfElements == a.Length
     else
       numberOfElements ==
         if front > rear
         then front - rear
         else front - rear + a.Length
 }
```

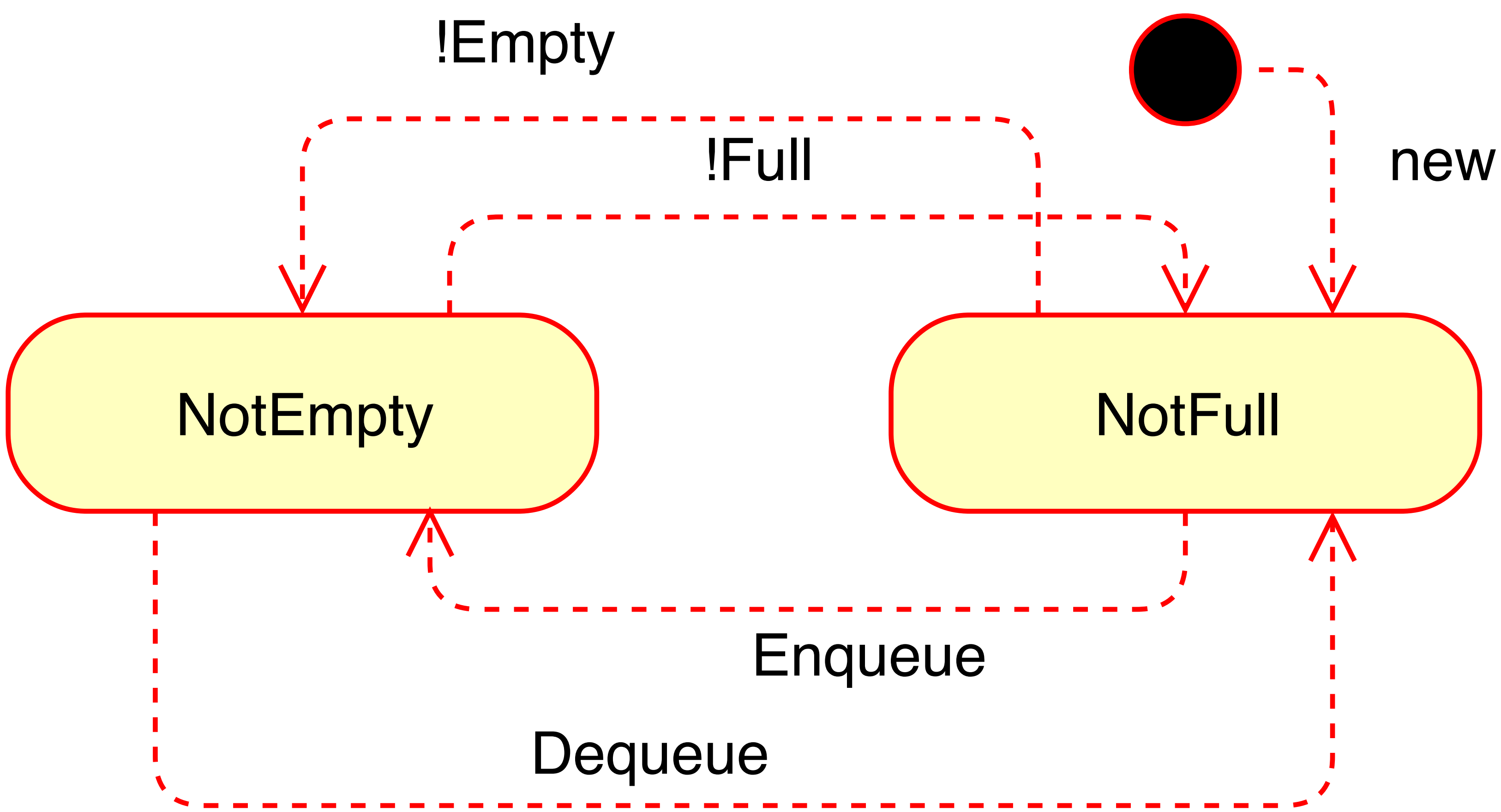
TypeStates - Queue

- Enqueue and Dequeue Operations are only valid in certain states… Obtained by dynamic testing operations.



TypeStates - Queue

- Enqueue and Dequeue Operations are only valid in certain states… Obtained by dynamic testing operations.



TypeStates - Queue

```
class Queue {
    …
    function NotFull():bool
        reads this
    { RepInv() && numberOfElements < a.Length }

    function NotEmpty():bool
        reads this
    { RepInv() && numberOfElements > 0 }

    constructor(N:int)
        requires 0 < N
        ensures NotFull()
        ensures fresh(a)
    { … }

    method Enqueue(V:int)
        modifies this`front, this`numberOfElements, a
        requires NotFull()
        ensures NotEmpty()
    { … }

    method Dequeue() returns (V:int)
        modifies this`rear, this`numberOfElements, a
        requires NotEmpty()
        ensures NotFull()
    { … }
```

```
method Main()
{
    var q:Queue := new Queue(4);
    var r:int;

    q.Enqueue(1);
    r := q.Dequeue();
    r := q.Dequeue();
    q.Enqueue(2);
    q.Enqueue(3);
    q.Enqueue(4);
    q.Enqueue(5);
}
```

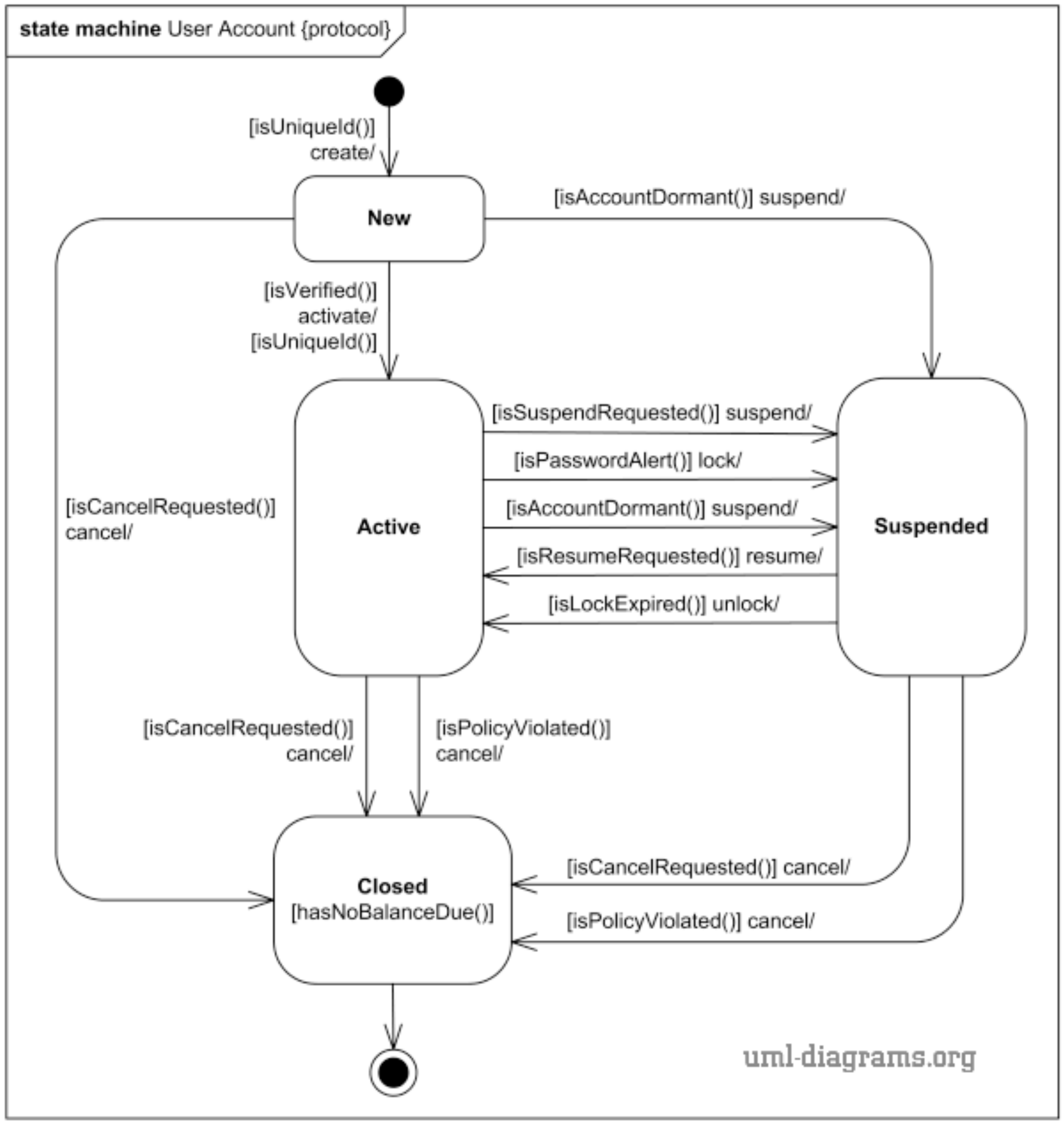
TypeStates - Queue

- Dynamic Tests ensure the proper state for a given operation…

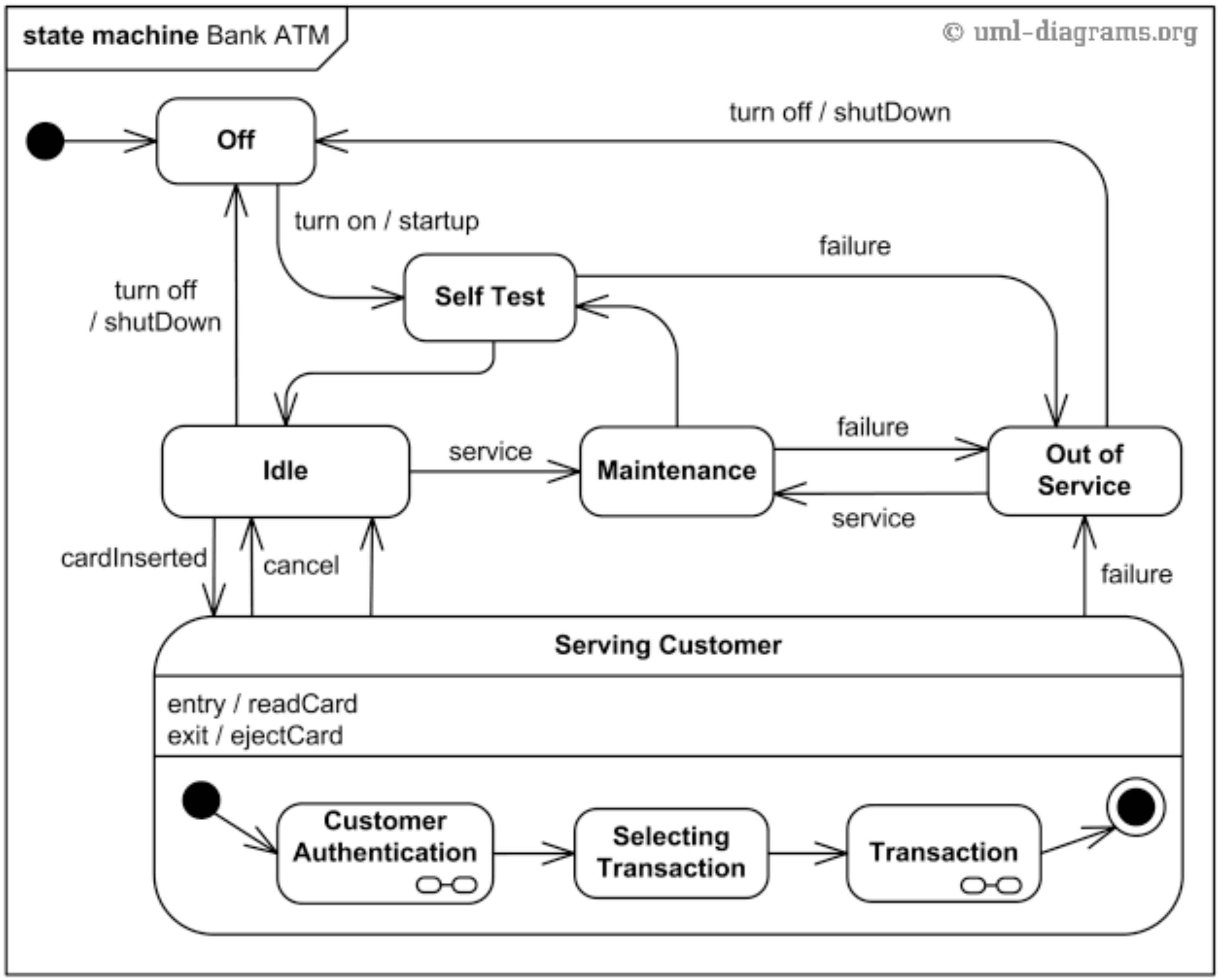
```
method Main()
{
    var q:Queue := new Queue(4);
    var r:int;

    q.Enqueue(1);
    r := q.Dequeue();
    if !q.Empty()
    { r := q.Dequeue(); q.Enqueue(2); }
    if !q.Full() { q.Enqueue(3); }
    if !q.Full() { q.Enqueue(4); r := q.Dequeue(); }
    if !q.Full() { q.Enqueue(5); }
}
```

TypeStates - UserAccount in a store



TypeStates - ATM



Further Reading

- **Program Development in Java**, ***Barbara Liskov and John Guttag***, Addison Wesley, 2003, Chapter 5 “Data Abstraction” (other book chapters are also interesting).

- **Programming with abstract data types**, ***Barbara Liskov and Stephen Zilles***, ACM SIGPLAN symposium on Very high level languages, 1974 (read the introductory parts, the rest is already outdated, but the intro is a brilliant motivation to the idea of ADTs). You can access this here: http://dl.acm.org/citation.cfm?id=807045.